Self-Adaptive Component-Based Transaction Commit Management

Patricia Serrano-Alvarado, Romain Rouvoy, Philippe Merle

Self-Adaptive Component-Based Transaction Commit Management

Patricia Serrano-Alvarado, Romain Rouvoy, Philippe Merle
INRIA Futurs – Jacquard Project,
Laboratoire d'Informatique Fondamentale de Lille,
UMR 8022 CNRS U.F.R. I.E.E.A. Bâtiment M3,
Université des Sciences et Technologies de Lille,
59655 Villeneuve d'Ascq Cedex, France.
{patricia.serrano-alvarado,romain.rouvoy,philippe.merle}@lifl.fr

ABSTRACT

For years, transactional protocols have been defined for particular application needs. Traditionally, when implementing a transaction service, a protocol is chosen and it remains the same during the system execution. Nevertheless, the dynamic nature of nowadays application contexts (e.g., mobile, ad-hoc, peer-to-peer) and behaviour variations (semantic-related aspects) motivates the needs for application adaptation. Next generation of system applications should be adaptive or even better self-adaptive. This paper proposes (1) a component-based architecture of standard 2PC-based protocols and (2) a self-Adaptive Component-based cOmmit Management, named ACOM. Self-adaptation is obtained by behaviour awareness and component-based reconfiguration. This allows ACOM to select the most appropriate protocol according to the context. We show that using ACOM performs better than using only one commit protocol in a variable system and that the reconfiguration cost can be negligible.

Keywords

Non-functional services, transaction management, commit protocols, component-based systems, self-adaptive systems.

1. INTRODUCTION

The dynamic nature of nowadays application contexts (e.g., mobile, ad-hoc, peer-to-peer) and behaviour variations (semantic-related aspects) motivates the needs for application adaptation. Next generation of applications should automatically tune themselves and apply optimizations in order to maximize performances, to evolve, to face different contexts or to adapt the execution process according to behaviour variations.

Component-based models are a good solution to make possible system adaptability [1]. This is because component-based architectures facilitate static and dynamic configuration. Implementing component-based adaptive applications is a very active and consolidated research/industrial issue, see some conference papers: [11, 4, 9]. Nevertheless, there has been little work on adaptability of non-functional services (e.g., persistence, replication, transaction), see these workshop papers: [8, 3].

In distributed transaction management, commit protocols are a key process. They ensure that all transaction operations success (commit) or none of them (abort). The most used commit protocol is the Two-Phase Commit (2PC) [7]. There exists a number of 2PC optimizations and some of them are so widely used that, as 2PC, are part of transaction processing standards. 2PC variations are proposed to optimize transaction execution costs, to address particular transaction semantics (e.g., read-only), to execute on different network topologies, etc. For instance, the 2PC Presumed Commit protocol (2PC-PC) [10] is well suited for high transaction commit rates, whereas 2PC Presumed Abort (2PC-PA) [10] is more appropriate for high transaction abort rates.

Traditionally, transaction service implementations are tailored for particular application behaviour. Transactional protocols are chosen and remain the same when the application behaviour changes. This may lead to unexpected poor performances. To deal with behaviour variations of transactional applications, the transaction management system should be adaptive or even better self-adaptive. We consider self-adaptation as the ability of being aware of the application behaviour changes and the capacity of reacting to them. This paper proposes a self-Adaptive Component-based cOmmit Management, named ACOM. Self-adaptation is obtained by a behaviour aware mechanism and component-based reconfiguration. This proposal is integrated in GoTM [12], an open component-based software framework, which allows constructing transaction services. The implementation performance results show that using ACOM performs better than using only one commit protocol in a variable system and that the reconfiguration cost can be negligible.

This paper is organized as follows. Section 2 briefly introduces the atomic commit protocols used in this work. Section 3 presents our approach, a self-adaptive commit management, and Section 4 gives some implementation details and performance results. Finally, Section 5 presents some

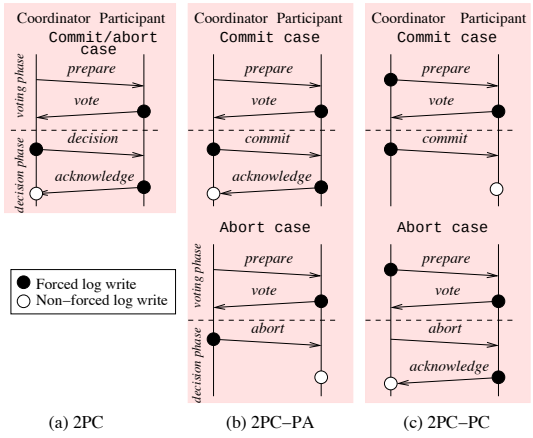


Figure 1: The 2PC, 2PC-PA and 2PC-PC protocols.

related work and Section 6 concludes and gives future work.

2. OVERVIEW OF COMMIT PROTOCOLS

In database systems, correct concurrent data access is ensured using transactions. Transactions are characterized by the well-known ACID (Atomicity, Consistency, Isolation and Durability) properties, which are guaranteed by transaction services. This work focuses on the atomicity property that is ensured by commit protocols.

For the purposes of this paper, we focus on some standard 2PC-based protocols. This section introduces 2PC, 2PC-PA and 2PC-PC (see Figure 1). It focuses on communication schema and logging issues. The resilience of commit protocols to system and communication failures is achieved by logging the progress of the protocol in the logs (stable storage) of the coordinator and the participants. There exist two types of log writes: *force* and *non-force*. The first one is immediately flushed into the log, generating a disk access. Non-force writes are eventually flushed into the log. Thus, there exists a window of vulnerability in using non-force writes until they are flushed.

2.1 Two-Phase Commit (2PC)

2PC, the most used commit protocol, consists of two phases (see Figure 1(a)). During the *voting phase*, the coordinator sends a *prepare* message to the participants. At the *decision phase*, the coordinator decides to commit (if all the participants vote *yes*) or abort (if at least one participant votes *no*) the transaction and notifies the participants of its decision. When the participants receive the final decision, they send an *acknowledge* message to the coordinator and release all resources held by the transaction. When the coordinator has received all the acknowledges from the participants that voted yes, it ends the protocol and forgets the transaction.

In 2PC, the coordinator force writes a *decision* record and non-force writes an *end* record at the end of the protocol. Participants force write their votes and the coordinator's decision. Write operations are logged before sending the related message.

Even though 2PC is widely implemented, it is considered as very expensive (see Table 1). It costs $4p$ message exchanges (being p the number of participants) and $1+2p$ forced log writes (the cost of non-forced log writes can be ignored).

Commit protocol	Messages		Forced log writes	
	Commit	Abort	Commit	Abort
2PC	$4p$		$1+2p$	
2PC-PA	$4p$	$3p$	$1+2p$	p
2PC-PC	$3p$	$4p$	$2+p$	$1+2p$

Table 1: The commit protocol costs.

2.2 2PC Presumed Abort (2PC-PA)

2PC-PA reduces the cost associated to aborted transactions. When the coordinator decides to abort a transaction, it discards all information about the transaction and sends an *abort* message to all the participants without logging the abort decision (see Figure 1(b) abort case). The participants non-force write the *abort* record and do not have to send an *acknowledge* message to the coordinator. In case of failures, the coordinator, not finding any information regarding the transaction will deduce an abort decision. The commit case of 2PC-PA remains the same as in 2PC.

Besides saving one force log write at the coordinator and at the participant's sites, 2PC-PA saves one *acknowledge* message from each participant for the abort case. Thus, the abort case of 2PC-PA costs $3p$ messages and p forced log write. The cost to commit a transaction is the same as in 2PC.

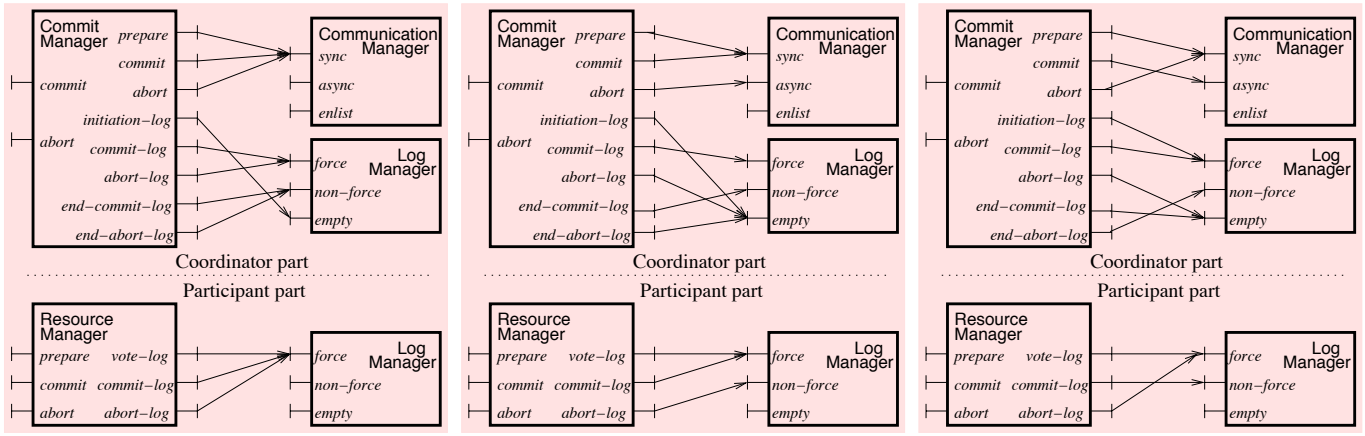
2.3 2PC Presumed Commit (2PC-PC)

2PC-PC, as opposed to 2PC-PA, reduces the cost of committed transactions. In 2PC-PC, the coordinator interprets missing information as a commit decision. To do so, the coordinator has to force write an *initiation* record for the transaction before sending *prepare* messages to participants (see Figure 1(c)). When the coordinator decides to commit a transaction it force writes a *commit* record then it sends the commit decision. The participants non-force write the commit decision and release all the transaction resources without acknowledging the commit decision to the coordinator. Otherwise, when the coordinator decides to abort a transaction, it sends *abort* messages to all the participants that voted yes and waits for the acknowledges. The abort decision is not logged. When all the acknowledges have been received, the coordinator writes a non-forced *end* record and discards all information pertaining to the transaction. The participants force write the abort decision and send an acknowledge to the coordinator.

Compared to 2PC, 2PC-PC saves one forced log write and one *acknowledge* message from each participant for the commit case at the expense of one extra *initiation* forced log write at the coordinator. Thus the cost of committing a transaction is $2+p$ forced log writes and $3p$ messages. For the abort case, 2PC-PC has one extra forced log write at the coordinator, the *initiation* record. Thus, aborting a transaction costs the same as in 2PC ($1+2p$ forced log writes and $4p$ messages).

2.4 Analysis

These protocols differ in the number of sent messages and the number of forced and non-forced log writes (Table 1 summarizes the commit protocol costs). These differences lead to different completion time of the commit processing, communication and disk access costs. Thus, it is cheaper to use 2PC-PA in a system where transactions are most likely going to abort, whereas, it is cheaper to use 2PC-



(a) 2PC protocol (b) 2PC–PA protocol (c) 2PC–PC protocol

Figure 2: Architecture of the commit protocols.

PC if transactions are most probably going to commit. In a system where transactions have the same probability of abort and commit, it is cheaper to use 2PC-PA.

3. ACOM

In this section we introduce our proposal, a self-Adaptive Component-based cOmmit Management (ACOM), which supports application behaviour variations. We argue that using component-based technologies facilitates static software configurations and dynamic reconfigurations. The rest of this paper validates this argument.

Section 3.1 introduces the architecture used to implement the commit protocols presented previously. Section 3.2 shows the behaviour aware mechanism used in our approach. Section 3.3 introduces the reconfiguration process. Finally, Section 3.4 discusses several issues concerning ACOM.

3.1 Architecture

Before getting into the architecture details, we extend the definition proposed in [13]. A component architecture (or *configuration*) is mainly composed of *components* and *bindings*. A component is a software entity, which exports functions through *server interfaces* and imports its dependencies via *client interfaces*. A binding connects a client interface to a server interface to resolve a component dependency.

The proposed architecture generalizes the commit protocols to reuse common functionalities. The objective is (1) to make a component-based implementation of the three commit protocols presented in Section 2 and (2) to express principal differences only through bindings. Indeed, each protocol reuses exactly the same components but with different configuration (see Figure 2).

From Figure 1, we identify four principal functionalities of the commit protocols: coordinator, participant, message exchange and logging. Thus, 2PC, 2PC-PA and 2PC-PC protocols are implemented reusing four components. The CommitManager component acts as the coordinator. It sends the *prepare*, *commit* and *abort* messages to all participants, which are represented by ResourceManagers, and logs the steps of the protocol. The CommunicationManager component, implemented as an event bus, allows the coordinator to send asynchronous or synchronous (using a callback approach) messages. There exists a LogManager component for the coordinator and for each participant. This component provides force and non-force log writes.

The coordinator part of this architecture is embedded in the transaction component whereas the participant part is implemented by resource managers (e.g., database managers) involved in the system.

2PC. In 2PC (Figure 2(a)), the CommitManager sends a synchronous *prepare* message. This means that the participants should attach their vote to the callback message returned to the coordinator. When a decision is taken, the CommitManager calls the *commit-log* (resp. *abort-log*) interface, which is connected to the *force* interface of the LogManager. The *commit* (resp. *abort*) message is sent synchronously to allow participants to *acknowledge* the decision. To terminate the protocol, the CommitManager non-force writes an *end* record calling the *end-commit-log* (resp. *end-abort-log*) interface. The ResourceManager receives (from the CommunicationManager) the *prepare* and *decision* messages. It force writes its vote and the coordinator's decision.

2PC-PA. In 2PC-PA (Figure 2(b)), as the abort decision is not logged, the *abort-log* interface is bound to the *empty* interface of the LogManager. This leaves the CommitManager code unchanged. Next, the *abort* message is sent asynchronously because the abort decision does not need to be acknowledged. Finally, the *end-abort-log* interface is connected to the *empty* LogManager interface because the end of an aborted transaction does not need to be logged. The commit case is the same as in 2PC. In the ResourceManager component, the commit case remains the same as in 2PC. In the abort case, *abort-log* is connected to the *non-force* interface of the participant's LogManager component.

2PC-PC. In 2PC-PC (Figure 2(c)), before sending the *prepare* message, the CommitManager calls the *initiation-log* interface. Compared to the other protocols, such an interface is connected to the *force* LogManager interface. In 2PC-PC, the commit decision is sent asynchronously because it is not necessary to acknowledge the commit decision. Since the

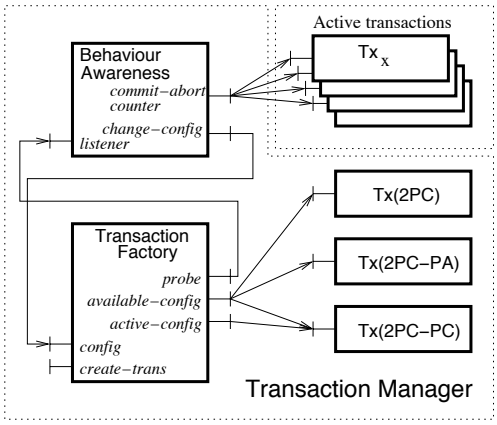


Figure 3: Behaviour awareness

end of a committed transaction does not need to be logged, the *end-commit-log* interface is connected to the *empty* interface of the LogManager. The abort decision is not logged, nevertheless, the *abort* message is sent synchronously. The end of an aborted transaction is non-force written into the log. In the ResourceManager component, the *commit-log* and *abort-log* interfaces are connected respectively to the *non-force* and *force* interfaces of the LogManager.

3.2 Behaviour awareness

In this paper, we consider the transaction abort and commit rate as the application behaviour. Thus, to be able of changing, at the right moment, the commit protocol it is necessary to monitor the abort and commit rates of transactions. This logic is named *adaptation policy*. An adaptation policy is defined by a kind of ECA rules (Event, Condition, Action). The Event is the commit/abort rate, the Condition specifies when it is necessary to change the active protocol and the Action is the protocol change.

The BehaviourAwareness component (see Figure 3) implements the adaptation policy. It monitors the number of committed and aborted transactions. Besides, it decides when the active protocol should be changed. This is possible thanks to the predefined ECA rules. For instance a ECA rule may say: *if abort-rate < 10% then use 2PC*.

To count the number of committed and aborted transactions, the BehaviourAwareness component uses the *enlist* interface provided by the CommunicationManager component of each transaction (see Figure 2). The *enlist* interface allows subscribing to different kind of events. Thus, the CommunicationManagers of all the active transactions notify the BehaviourAwareness component when the CommitManagers send *commit* and *abort* messages to the participants.

3.3 Reconfiguration

Knowing the current commit/abort rate allows predicting the future transaction behaviour. That is, if the abort rate is about 30%, we consider that this tendency will remain the same in a near future. This is why the abort/commit rate motivates the reconfiguration.

When the BehaviourAwareness component decides to change a protocol (based on defined Conditions) it calls the *change-config* interface connected to the *config* interface of the TransactionFactory component. Then the TransactionFactory component connects the *active-config* interface to the appropriate configuration, which is listed by the *available-config* interface. Thus, next transactions are created using this new active configuration. In Figure 3, we show the transaction implementation containing the three commit protocols (Tx(2PC), Tx(2PC-PC) and Tx(2PC-PA)). The active configuration is Tx(2PC-PC).

When TransactionFactory component creates a new transaction, it enlists the *listener* interface (retrieved via the *probe* interface) of the BehaviourAwareness component to make possible the commit/abort event monitoring.

3.4 Discussion

This section discusses various general and different aspects concerning ACOM.

Reconfiguration of active transactions. Changing the protocol of active transactions compromises the recovery process in case of failures. That is why in ACOM it is not possible to change the commit protocol once a transaction has begun. Different active transactions can use different commit protocols but each transaction begins and ends with the same commit protocol.

Using ACOM. To be able of using ACOM, for instance, in an application server, the following hypotheses should be guaranted. 1) The participant part is implemented by resource managers that are free to choose the way this implementation is done (Figure 2 suggests one implementation solution); 2) All considered protocols in ACOM must be implemented by resource managers; finally, 3) Resource managers must be able to change the active protocol.

ACOM extension. ACOM may support other commit protocols that can be different to those used in this paper. We choose 2PC-based protocols as an experience to show components reusability. Nevertheless, reusability is not necessary to the ACOM operation. Thus, with ACOM it is possible to switch to different commit protocol implementations, which makes it extensible.

Preserving the global semantics of the system. In software reconfiguration, it is necessary to preserve the semantics of the system. In our case, the transaction properties must be preserved. If an atomic commit protocol is replaced by another, which does not enforce the atomicity property (for instance, the semantic atomicity [6]), the transaction correctness is compromised. This is why in this paper, used protocols ensure the atomicity property. Thus, programmers must be careful about the choices they made when defining adaptive middleware systems.

4. IMPLEMENTATION ISSUES

ACOM has been implemented in GoTM, which uses the Fractal component model. Next two sections introduce Fractal (4.1) and GoTM (4.2). Section 4.3 presents some performance results.

4.1 Fractal

Fractal [1] is a modular and extensible component model developed by INRIA and France Telecom. It is hosted by the ObjectWeb international consortium and it is used in both academic and industrial development projects. The choice of Fractal in this work is motivated by the fact that it provides

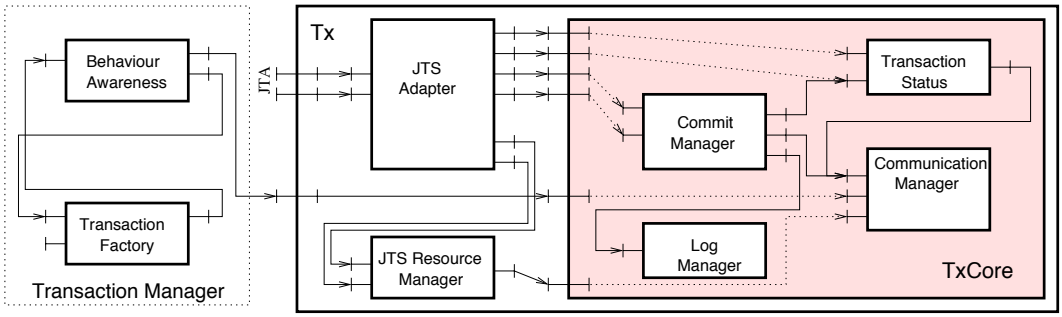


Figure 4: A GoTM implementation using ACOM.

good performance in both local and remote contexts [5] and because it allows dynamic reconfigurations.

Fractal distinguishes between two kinds of components: primitive and composite components. A primitive component is implemented in a given programming language. A composite component provides a means of dealing with a group of components as a whole.

Communication is performed through interfaces. There are two categories of interfaces, those provided by the component (server interfaces) and those required by the component (client interfaces).

Dynamic reconfiguration process is done atomically in accordance with the component life cycle (creation, activation, deactivation, destruction). Thus, the dynamic reconfiguration involves the deactivation, the reconfiguration and the reactivation of involved components. This means that reconfiguration operations can only be performed when the component is deactivated. This is, the component activities terminate and the incoming method calls are suspended until the component is reactivated.

Finally, Fractal provides an Architecture Description Language (ADL) to describe and deploy automatically component-based configurations.

4.2 GoTM

GoTM [12], like Fractal, is a project developed as part of the ObjectWeb initiative. It is a component-based software framework, developed in Java, that allows constructing adaptable transaction services. It proposes a solution to deal with heterogeneity of existing transaction services and standards. The idea is to provide common transaction functionalities in a core layer and an adaptation layer addressed to different transaction standards (e.g., CORBA and Java Transaction services).

GoTM provides a set of Fractal components implementing generic transaction-related functions and strategies. The static configuration of the transaction service is described using the Fractal ADL, which allows the transaction service designer to select the default strategies to use.

The GoTM framework includes different optimizations to provide good performance. They include the use of a pool of components to reduce the cost of component creation. GoTM uses configurable factories to describe and configure created component instances. Threading strategies (e.g., sequential, threaded, or pooled) control the propagation of messages (synchronous or asynchronous).

Figure 4 shows the general architecture of a GoTM implementation that supports JTS transactions [2]. The Tx component represents a JTS transaction where the TxCore component groups the core functions provided by GoTM. Core components include those presented in Figure 2 plus other general components such as the TransactionStatus. Figure 4 shows the TransactionManagement component and its relation with the Tx's CommunicationManager component.

The commit protocol reconfiguration is done through a dedicated attribute. This attribute is read by the TransactionFactory component when new transactions are created. Thus, the reconfiguration process consists of changing the value of this attribute depending on the predefined Conditions.

4.3 Performance results

The objective of this section is (1) to confirm the presumptions made in Section 2 about 2PC, 2PC-PC and 2PC-PA, and (2) to highlight the performance of our proposal (ACOM).

The scenario of Figures 5 and 6 evaluates the average completion time of a number of transactions executed sequentially varying the number of accessed resources (from 0 to 20). This scenario is applied to the 2PC, 2PC-PA and 2PC-PC protocols.

In Figure 5, all executed transactions are aborted. This shows that 2PC-PA performs much better than 2PC and 2PC-PC. This is because 2PC-PA saves one acknowledge message from each participant in the abort case (see Section 2.2). 2PC and 2PC-PC have similar performance because they have similar costs even if they differ in the phase in which the coordinator makes a force log write (see Section 2.3).

In Figure 6, all executed transactions are committed. In this case, 2PC-PC behaves better than 2PC and 2PC-PA. This is because 2PC-PC saves one force log write and one acknowledge message from each participant. 2PC and 2PC-PA have similar performance because their commit case follows the same process.

The scenario of Figure 7 evaluates the average completion

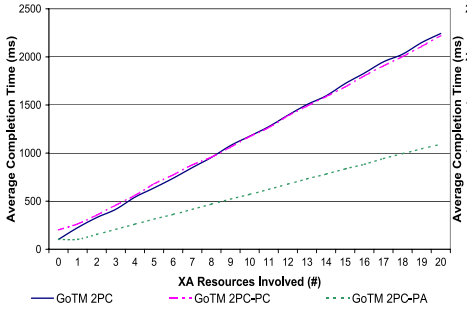


Figure 5: Performance with high abort rates.

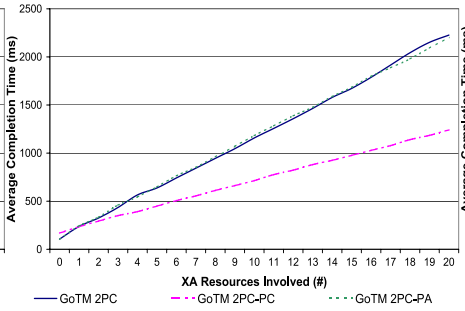


Figure 6: Performance with high commit rates.

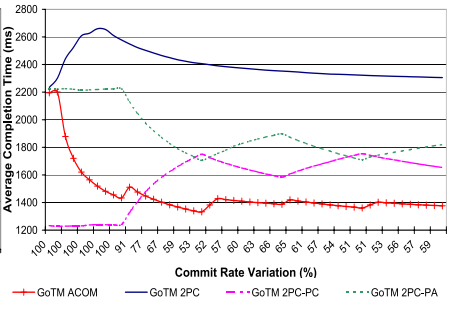


Figure 7: Performance with variable commit/abort rates.

time of 100 transactions executed sequentially with constant commit/abort rate variations (20 transactions commit, then 20 transactions abort, then 20 transactions commit, etc.). 2PC, 2PC-PA and 2PC-PC are executed and compared to ACOM. The Condition that specifies when to change of commit protocol in ACOM (see Section 3.2) is based on the following equation:

$$\begin{cases} x + y = 100 \\ x \times C_{2PC-PC} + y \times A_{2PC-PC} < x \times C_{2PC-PA} + y \times A_{2PC-PA} \end{cases}$$

Where x (resp. y) represents the number of transaction committed (resp. aborted) and C_{2PC-PX} (resp. A_{2PC-PX}) represents the commit (resp. abort) cost of the 2PC-PX protocol (2PC-PA and 2PC-PC).
The solution of this equation is:

$$\begin{cases} y = 100 - x \\ x > \frac{100 \times (A_{2PC-PA} - A_{2PC-PC})}{(C_{2PC-PC} - A_{2PC-PC} - C_{2PC-PA} + A_{2PC-PA})} \end{cases}$$

When applying this solution to the measures of Figure 5 and 6, 2PC-PC becomes more interesting than 2PC-PA when the commit rate is above 54%. This limit is used by ACOM to switch between the 2PC-PC and the 2PC-PA configurations.

The measures of Figure 7 show the average completion time that varies depending on the transaction commit/abort rates. Performance of ACOM is the best thanks to its capacity of self-adaptation. 2PC-PA and 2PC-PC suffer from the behaviour variations. In ACOM, when the commit rate is high (in this experience, 54%), the active protocol is 2PC-PC. Otherwise, ACOM uses 2PC-PA. Thus, ACOM benefits of the best performance of 2PC-PC and 2PC-PA. In this experience, 2PC is used as the initial protocol. ACOM does not switch to 2PC because taking as behaviour only the commit/abort rate, 2PC is more expensive than the other considered protocols.

Finally, performances of Figure 7 show that the ACOM reconfiguration does not introduce important overheads compared to the static configuration of the use cases protocols.

5. RELATED WORK

[3] proposes to dynamically adapt applications by composing at runtime (by weaving) functional (application-related) and non-functional concerns. Authors are interested in making the weaving process adaptive to runtime execution conditions. Their objective is to choose at runtime the appropriate non functional code. Thus, they propose to change the weaving of non-functional code according to context aware adaptation policies.

[8] proposes runtime application adaptability by assembling appropriate non-functional services thanks to service repositories. Repositories contain component-based non-functional services and meta-information describing such services. This approach requires the applications to be developed using the component-based approach. Our approach does not make any assumption about the application design and we choose to adapt the non-functional service itself rather than the instance of used service.

Compared to our proposal, [3] and [8] consider non-functional services as the adaptation grain. Our approach proposes self-adaptability of non-functional services using components as adaptation granularity. Unlike [3] and [8], we made several experiences that underline the advantages of our proposal.

[14] proposes a new commit protocol for self-adaptive Web services, which supports both 2PC-PA and 2PC-PC participants. Such a protocol allows participants with different presumptions to be dynamically combined in one transaction. Compared to the work presented in this paper, [14] does not address evolution concerns. In our work, we use 2PC, 2PC-PA and 2PC-PC as use cases. Our approach can easily support new commit protocols to extend the application adaptive ability.

6. CONCLUSIONS AND FUTURE WORK

Self-adaptation is a current challenge in component-based software engineering. Several works have been devoted to adaptive applications, nevertheless, there has been little work on adaptability of non-functional services. This paper focused on transaction services, in particular, on the commit process. On the one hand, it proposed a component-based architecture of standard 2PC-based protocols. Each protocol contains exactly the same components but with different configurations. On the other hand, it proposed a self-adaptive component-based commit management (ACOM). Performance measures show that changing the commit protocol depending on the behaviour context performs better that using only one commit protocol on a variable transactional system.

Our future work includes to study the component-based con-

figuration of other 2PC-based protocols (e.g., [14]) but also 1PC and 3PC protocols. The idea is to extend GoTM to support more commit protocols. The evaluation of runtime performances of these new components will be useful to refine the ACOM adaptation policies, e.g., adding new conditions and reconfiguration actions to switch between protocols.

Besides, we consider to investigate a model-driven approach to design commit protocols (e.g., using UML sequence diagrams) and to automatically generate the implementation of the `CommitManager` components and their bindings to the `CommunicationManager` and the `LogManager` components. This model-driven approach, complementary to that defined into [12], will provide a dedicated high level language to define, study, compare commit protocols, and also an efficient way to implement them with GoTM.

7. REFERENCES

[1] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J. Stefani. An Open Component Model and its Support in Java. In *Int. Symp. on Component-Based Software Engineering (CBSE)*, Edinburgh, UK, May 2004.

[2] S. Cheung. *Java Transaction Service Specification*. Sun Microsystems Inc., San Antonio Road, Palo Alto, CA, version 1.0 edition, December 1999.

[3] P. David and T. Ledoux. Dynamic Adaptation of Non-Functional Concerns. In *ECOOP Workshop on Unanticipated Software Engineering*, Malaga, Spain, June 2002.

[4] P. David and T. Ledoux. Towards a Framework for Self-Adaptive Component-Based Applications. In *Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*, November 2003.

[5] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. *Studia Informatica Universalis Regular Issue*, 4(1), 2005.

[6] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems (TODS)*, 8(2), 1983.

[7] J. Gray. Notes on Database Operating Systems. In *Advanced Course: Operating Systems*, number 60 in LNCS, 1978.

[8] C. Hérault and S. Lecomte. Gestion Dynamique des Services Techniques pour Modèle à Composants. In *Conf. Francophone sur le Déploiement et la (Re) Configuration de Logiciels (DECOR)*, Grenoble, France, October 2004.

[9] O. Layaida and D. Hagimont. Designing Self-Adaptive Multimedia Applications trough Hierarchical Reconfiguration. In *Int. Conf. on on Distributed Applications and Interoperable Systems (DAIS)*, Athens, Greece, June 2005.

[10] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems (TODS)*, 11(4), 1986.

[11] A. Rasche, M. Puhlmann, and A. Polze. Heterogeneous Adaptive Component-Based Applications with Adaptive.Net. In *Int. Symp. on Object-Oriented Real Time Distributed Systems*, May 2005.

[12] R. Rouvoy and P. Merle. Towards a Model Driven Approach to Build Component-Based Adaptable Middleware. In *Workshop on Reflective and Adaptive Middleware (RAM 2004)*, Toronto, Canada, October 2004.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[14] W. Yu, Y. Wang, and C. Pu. A Dynamic Two-Phase Commit Protocol for Self-Adapting Services. In *Int. Conf. on Services Computing (SCC)*, 2004.